Lecture 10 — Shortest Paths II (DRAFT)

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2011)

Lectured by Guy Blelloch — September 29, 2011

Today:
- Continuation of Dijkstra's algorithm (see notes from previous lecture)
- Bellman-Ford's Algorithm, which allows negative weights

1 The Bellman Ford Algorithm

We now turn to solving the single source shortest path problem in the general case where we allow negative weights in the graph. One might ask how negative weights make sense. If talking about distances on a map, they probably do not, but various other problems reduce to shortest paths, and in these reductions negative weights show up. Before proceeding we note that if there is a negative weight cycle (the sum of weights on the cycle is negative), then there cannot be a solution. This is because every time we go around the cycle we get a shorter path, so to find a shortest path we would just go around forever. In the case that a negative weight cycle can be reached from the source vertex, we would like solutions to the SSSP problem to return some indicator that such a cycle exists and terminate.

Exercise 1. *Consider the following* currency exchange *problem: given the a set currencies, a set of exchange rates between them, and a source currency s, find for each other currency v the best sequence of exchanges to get from s to v. Hint: how can you convert multiplication to addition.*

Exercise 2. *In your solution to the previous exercise can you get negative weight cycles? If so, what does this mean?*

So why is it that Dijkstra's algorithm does not work with negative edges? What is it in the proof of correctness that fails? Consider the following very simple example:

```
   a o
     / \
  3 /   \ -2
s o --- o b
      2
```

Dijkstra's algorithm would visit b then a and leave b with a distance of 2 instead of the correct -1. The problem is that it is no longer the case that if we consider the closest vertex not in the visited set that it needs to have a path through the visited set.

So how can we find shortest paths on a graph with negative weights. As with most algorithms, we should think of some inductive hypothesis. In Dijkstra, the hypothesis was that if we have found the

i nearest neighbors, then we can add one more to find the $i + 1$ nearest neighbors. Unfortunately, as discussed, this does not work with negative weights, at least not in a simple way.

What other things can we try inductively. There are not too many choices. We could think about adding the vertices one by one in an arbitrary order. Perhaps we could show that if we have solved the problem for i vertices then we can add one more along with its edges and fix up the graph cheaply to get a solution for $i + 1$ vertices. Unfortunately, this does not seem to work. Similarly doing induction on the number of edges does not seem to work. You should think through these ideas and figure out why they don't work.

How about induction on the unweighted path length (from now on we will refer to path length as the number of edges in the path, and path weight as the sum of the weights on the edges in the path). In particular the idea based on induction is that, given the shortest weighted path of length at most i (i.e. involving at most i edges) from s to all vertices, then we can figure out the shorted weighted path of length at most $i + 1$ to all vertices. It turns out that this idea does pan out, unlike the others. Here is an example:

```
     i  1  i
   a o --- o c
    / \-2   \ 1       Closest distances from s for paths of length 0
 3 /   \ b   \               i indicates infinity
s o --- o --- o d
0    2  i 1   i

     3  1  i
   a o --- o c
    / \-2   \ 1       for paths of length 1
 3 /   \ b   \
s o --- o --- o d
0    2  2 1   i

     0  1  4
   a o --- o c
    / \-2   \ 1       for paths of length 2
 3 /   \ b   \
s o --- o --- o d
0    2  2 1   3

     0  1  1
   a o --- o c
    / \-2   \ 1       for paths of length 3
 3 /   \ b   \
s o --- o --- o d
0    2  2 1   3

     0  1  1
   a o --- o c
    / \-2   \ 1       for paths of length 4
 3 /   \ b   \
s o --- o --- o d
0    2  2 1   2
```

Here is an outline of a proof that this idea works by induction. This proof also leads to an algorithm. We use the convention that a vertex that is not reachable with a path length i has distance infinity (∞) and set the initial distance to all vertices to ∞. For the base case, on step zero no vertices except for the source are reachable with path length 0, and the distance to all such vertices is ∞. The distance to the source is zero. For the inductive case we note that any path of length $i+1$ has to go through a path of length i plus one additional edge. Therefore we can figure out the shortest length $i+1$ path to v by considering all the in-neighbors $u \in N_G^-(v)$ and taking the minimum of $w(u,v) + d(u)$.

Here is the Bellman Ford algorithm based on this idea. The notation $\delta_G^i(s,v)$ indicates the shortest path from s to v in G that uses at most i edges.

```
1   % implements: the SSSP problem
2   fun BellmanFord($G = (V,E), s$) =
3   let
4      % requires: all$\{D_v = \delta_G^i(s,v) : v \in V\}$
5      fun $BF(D,i)$ =
6         let
7            val $D' = \{v \mapsto \min_{u \in N_G^-(v)}(D_u + w(u,v)) : v \in V\}$
8         in
9            if $(i = |V|)$ then $\perp$
10           else if (all$\{D_v = D'_v : v \in V\}$) then $D$
11           else $BF(D', i+1)$
12        end
13     val $D = \{v \mapsto$ if $v = s$ then $0$ else $\infty : v \in V\}$
14  in $BF(D,0)$ end
```

In Line 9 the algorithm returns $\perp$ if there is a negative weight cycle. In particular since no simple path can be longer than $|V|$ if the distance is still changing after $|V|$ rounds, then there must be a negative weight cycle.

We can analyze the cost of the algorithm. First we assume the graph is represented as a table of tables, as we suggested for the implementation of Dijkstra. We then consider representing it as a sequence of sequences. We use the following cost table. The reduce complexity is assuming the combining function takes constant work.

	Work	*Span*
Array Sequence		
`tabulate` $f\ n$	$O\left(\sum_{i\in[0,n)} W(f(i))\right)$	$O\left(\max_{i\in[0,n)} S(f(i))\right)$
`reduce`* $f\ v\ S$	$O(\|S\|)$	$O(\log\|S\|)$
`nth` $S\ i$	$O(1)$	$O(1)$
Tree Table		
`tabulate` $f\ T$	$O\left(\sum_{k\in S} W(f(k))\right)$	$O\left(\log\|T\| + \max_{k\in S} S(f(k))\right)$
`reduce`* $f\ v\ T$	$O(\|T\|)$	$O(\log\|T\|)$
`find` $T\ k$	$O(\log\|T\|)$	$O(\log\|T\|)$

Cost of Bellman Ford using a Tables Here we assume the graph G is represented as a $(\mathbb{R}$ `vTable`$)$ `vTable`, where `vTable` maps vertices to values. The $\mathbb{R}$ are the real valued weights on the edges. We assume the distances D are represented as a $\mathbb{R}$ `vTable`. Lets consider the cost of one call to BF, not including the recursive calls. The only non trivial computations are on lines 7 and 10. Line 7 consists of a tabulate over the vertices. As the table indicates, to calculated the work we take the sum of the work for each vertex, and for the span we take the maximum of the spans, and add $O(\log n)$. Now consider what the algorithm does for each vertex. First it has to find the neighbors in the graph (using a `find G v`). This requires $O(\log |V|)$ work and span. Then it involves a map over the neighbors. Each instance of this map requires a find in the distance table to get D_u and an addition of the weight. The find takes $O(\log |V|)$ work and span. Finally there is a reduce that takes $O(1 + |N_G(v)|$ work and $O(\log |N_G(v))|$ span. Using $n = |V|$ and $m = |E|$, the overall work and span are therefore

$$\begin{aligned}
W &= O\left(\sum_{v\in V}\left(\log n + |N_G(v)| + \sum_{u\in N_G(v)}(1+\log n)\right)\right)\\
&= O\left((n+m)\log n\right)\\
S &= O\left(\max_{v\in V}\left(\log n + \log|N_G(v)| + \max_{u\in N(v)}(1+\log n)\right)\right)\\
&= O(\log n)
\end{aligned}$$

Line 10 is simpler to analyze since it only involves a tabulate and a reduction. It requires $O(n\log n)$ work and $O(\log n)$ span.

Now the number of calls to BF is bounded by n, as discussed earlier. These calls are done sequentially so we can take multiply the work and span for each call by the number of calls giving:

$$\begin{aligned}
W(n,m) &= O(nm\log n)\\
S(n,m) &= O(n\log n)
\end{aligned}$$

Cost of Bellman Ford using Sequences If we assume the vertices are the integers $\{0, 1, \ldots, |V| - 1\}$ then we can use array sequences to implement a `vTable`. Instead of using a `find` which requires $O(\log n)$ work, we can use `nth` requiring only $O(1)$ work. This improvement in costs can be applied for looking up in the graph to find the neighbors of a vertex, and looking up in the distance table to find the current distance. By using the improved costs we get:

$$\begin{aligned} W &= O\left(\sum_{v\in V}\left(1 + |N_G(v)| + \sum_{u\in N_G(v)} 1\right)\right) \\ &= O(m) \\ S &= O\left(\max_{v\in V}\left(1 + \log|N_G(v)| + \max_{u\in N(v)} 1\right)\right) \\ &= O(\log n) \end{aligned}$$

and hence the overall complexity for Bellman Ford with array sequences is:

$$\begin{aligned} W(n,m) &= O(nm) \\ S(n,m) &= O(n\log n) \end{aligned}$$

By using array sequences we have reduced the work by a $O(\log n)$ factor.